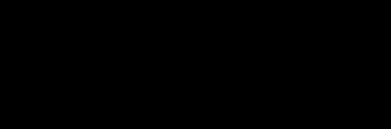


Game Solutions: Graphics and Events

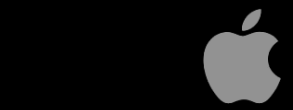
Session 511

Game Solutions: Graphics and Events

David Hill and Todd Previte
Developer Technical Support

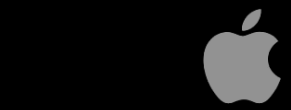
Introduction

- Unique aspects of Mac OS X game development
- Un(der)documented areas
- Tips and tricks
- Concepts new to Mac OS 9 developers



What You Will Learn

- **CGDirectDisplay** changes
- Full screen debugging techniques
- Carbon controls over OpenGL
- QuickTime and OpenGL
- OpenGL Buffer Ops
- Event loops and timers



CGDirectDisplay

What is new?

- Additional display mode dictionary flags
  - **kCGDisplayModeIsStretched**
- Mirroring
- Display configuration
- Capture all displays
- Fading to and from a solid color

Fullscreen Debugging

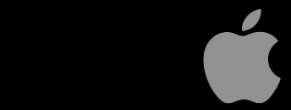
Typical scenario

- You have captured the screen and faded to black
- You are supposed to fade back in, but you do not
- What do you do?
- How do you tell what went wrong?

Fullscreen Debugging

Options

- **printf** and other logging
- **DrawSprocket** debug mode
- Remote debugging



Fullscreen Debugging

printf and other logging

- Good for post-mortem debugging
- Requires that you modify your code
- Allows for a “binary search” strategy
- Not terribly efficient

Fullscreen Debugging

DrawSprocket debug mode

- Great if you are using **DrawSprocket**
- Does not show blanking window
- Does not do complete fades
- Works on Mac OS 9 or X

Fullscreen Debugging

Enabling DrawSprocket debug mode

- Two ways to turn it on
  - Call **DSpSetDebugMode(true);**
  - Create a folder called "**DSpSetDebugMode**"
- Must enable before activating your context
- Must use the debug build of DrawSprocket

Fullscreen Debugging

Remote debugging

- Uses the power of Mac OS X
- Requires a second machine
- Requires **ssh** and **gdb**
- Does not provide an IDE
- Does allow for source-level debugging

Fullscreen Debugging

Getting help in gdb

- **help**
- **help commandClassName**
  - Lists **gdb** commands in that class
  - Examples: breakpoints, stack, etc.
- **help commandName**
  - Gives full documentation for the command

Fullscreen Debugging

Getting started in gdb

- attach pid
- file pathToAppToDebug
  - Watch out for bundled apps
  - AppDir/Contents/MacOS/AppToDebug
- run

Fullscreen Debugging

Controlling execution in gdb

- **step**—execute one line (step-into)
- **next**—execute one line (step-over)
- **finish**—finish stack frame (step-out)
- **continue**—continue execution (go)

Fullscreen Debugging

Setting breakpoints in gdb

- **b file:lineNumber** or **functionName**
- **fb file:lineNumber** or **functionName**
- **b**—by itself sets breakpoint at PC
- **clear addressOfBreakpoint**
- **delete breakpointNumber**
- **info breakpoints**

Fullscreen Debugging

Getting info once you have stopped

- **bt**—stack backtrace
- **info locals**—locals from the current stack frame
- **info variables**—all globals and statics
- **whatis variableName**—shows type
- **list**—shows source around PC
- **info source**—current source file info

Fullscreen Debugging

Displaying variables and memory in gdb

- **x/fmt address** or **variableName**
  - Repeat count
  - (o)ctal,he(x),(d)ecimal, (f)loat, (u)nsigned decimal, (c)har, (s)tring
  - (b)yte, (h)alfword, (w)ord, (g)iant
  - **x/f floatVariableName**
- **set variableName=newValue**
- **po objectiveCObjectRef**—calls description on object

Fullscreen Debugging

Calling functions in gdb

- **call (returnType)functionName(args)**
- **call (void)CFShow(CFTypeRef)**

- Extremely useful for custom diagnostics
  - **call (void)dumpMyInternalState(foo)**
- Watch out for side effects

Demo

Fullscreen Debugging

Controls Over OpenGL

Possible solutions

1. Implement your own controls
2. Place controls in a higher window
3. Place controls in an overlay window
4. Surface ordering

Controls Over OpenGL

1. Implement your own controls

- Pros
  - Precise control over drawing
  - Place onscreen with OpenGL
- Cons
  - Hard to mimic Aqua UE
  - Have to draw everything yourself

Controls Over OpenGL

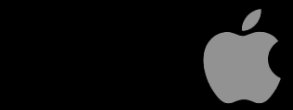
2. Place controls in a higher window

- Pros
  - Do not have to draw controls
  - Do not have to handle interaction
- Cons
  - Blocks out rectangular area
  - Multiple windows to manage

Controls Over OpenGL

3. Place controls in an overlay window

- Pros
  - OS handles controls
  - OpenGL content shows through
- Cons
  - Speed penalty with software compositor
  - Multiple windows to manage



Controls Over OpenGL

4. Surface ordering

- Pros
  - Works with both Cocoa and Carbon
  - OpenGL content shows through
- Cons
  - Speed penalty with software compositor
  - Requires Jaguar

Controls Over OpenGL

Overlay window steps

- Create the overlay window
- Create the window group
- Add main and overlay windows to the group
- Install your controls
- Install Carbon Event handlers
    - Handle control or command events

Controls Over OpenGL

- Create the overlay window

```
OSStatus status = CreateNewWindow (
            kOverlayWindowClass,
            kWindowNoAttributes,
            &wRect,
            &overlayWindowRef );
```

Controls Over OpenGL

- Create the window group

```
status = CreateWindowGroup (
                0, /* no group attributes */
                &windowGroupRef );

status = SetWindowGroupParent (
                windowGroupRef,
                GetWindowGroupOfClass(
                        kDocumentWindowClass ));
```

Controls Over OpenGL

- Add windows to the group

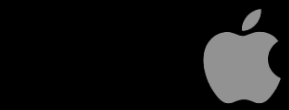
```
status = SetWindowGroup (
                openGLWindowRef,
                windowGroupRef );

status = SetWindowGroup (
                overlayWindowRef,
                windowGroupRef );
```

Controls Over OpenGL

- Set window group attributes

```
status = ChangeWindowGroupAttributes (
                    windowGroupRef,
                    kWindowGroupAttrMoveTogether |
                    kWindowGroupAttrLayerTogether |
                    kWindowGroupAttrHideOnCollapse,
                    0 /* don't clear any attributes */ );
```



Demo

Controls Over OpenGL

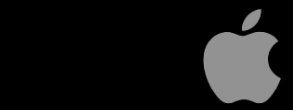
What You Will Learn

- **CGDirectDisplay** changes
- Full screen debugging techniques
- Carbon controls over OpenGL
- Event loops and timers
- QuickTime and OpenGL
- OpenGL Buffer Ops

Events and Timers

Carbon Events

- **RunApplicationEventLoop()**
  - Entry to Carbon Events
- **QuitApplicationEventLoop()**
  - Exits **RunApplicationEventLoop**
  - Called from your ‘quit’ handler
- **WaitNextEvent()**
  - WNE is dead, Use Carbon Events
  - WNE is just a shim layer on top of Carbon Events



Events and Timers

Carbon Events

- Event Levels
  - Application
    - Use for application-wide events
    - Highest level possible
  - Window
    - Use if you want to constrain events to a window
      - Especially important for mouse events
  - Menu
    - Must use if using standard menus
  - Control
    - Install a handler on a control directly

Events and Timers

Carbon Events

- Event Classes
  - Mouse
    - Button clicks
    - Motion
    - Motion deltas
  - Keyboard
    - Key up and down
    - Modifiers
  - Window
    - Activation and deactivation
    - Show and hide
  - Menu
    - Selection of menu items

Events and Timers

Handling Carbon Events

- Events and event lists

```
GetEventTypeCount(appEventList),

static const EventTypeSpec  appEventList[] = {
        {kEventClassCommand, kEventCommandProcess},
        {kEventClassMouse, kEventMouseDown},
        {kEventClassMouse, kEventMouseUp},
        {kEventClassMouse, kEventMouseMoved},
        {kEventClassMouse, kEventMouseDragged},
        {kEventClassMouse, kEventMouseWheelMoved}
};
```

Events and Timers

Handling Carbon Events

- Sample event information

{kEventClassCommand, kEventCommandProcess},

- Command processes
- Quit, OK, Cancel, Undo, Redo, etc.

{kEventClassMouse, kEventMouseDown},

- Sent when a mouse button is pressed
- Where, which button, etc.

{kEventClassMouse, kEventMouseMoved},

- Sent when the mouse moves
- Deltas, end point, etc.

Events and Timers

Handling Carbon Events

- Installing Standard Event Handlers

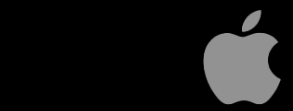
```
// Default handler for the primary window of the app
InstallStandardEventHandler(
                    GetWindowEventTarget(pMainWindow));
// Default handler for application-level events
InstallStandardEventHandler(
                    GetApplicationEventTarget());
// Default handler for the standard menu
InstallStandardEventHandler(
                    GetMenuEventTarget(theMenu));
```

Events and Timers

Handling Carbon Events

- Installing Event Handlers

```
// Register our UPP and the callback
appCommandProcessor = NewEventHandlerUPP(
                              MainAppEventHandler);
// Install the handler for the application
err = InstallApplicationEventHandler(
            appCommandProcessor,                  // Callback routine
            GetEventTypeCount(appEventList),      // Event Count
            appEventList,                         // Event list
            0,                                    // User data
            NULL);                                // EventHandlerRef
```



Events and Timers

Carbon Timers

- Timer Uses
  - Control of rendering loops
  - Handling application (game) events
  - Time-based events

Events and Timers

- Installation of Timers

```
// TimerRef to keep track of our timer
EventLoopTimerRef snapshotTimer;
// Timer universal proc pointer
EventLoopTimerUPP snapshotTimerUPP;
// The function prototype for the callback the timer will use
pascal void SnapshotTimedEventProcessor(
                    EventLoopTimerRef inTimer,
                    void* timeData);
```

Events and Timers

- Installation of Timers

```
// Set the timer UPP to point to our callback
snapshotTimerUPP =
NewEventLoopTimerUPP(SnapshotTimedEventProcessor);
// Install the timer
InstallEventLoopTimer(
        GetCurrentEventLoop(),                          // Where
        0,                                              // Start delay
        kEventDurationMillisecond * seqTimeInt,         // Interval
        snapshotTimerUPP,                               // Timer UPP
        NULL,                                           // User Data
        &snapshotTimer);                                // TimerRef
```

Events and Timers

- Removal of Timers

```
// First remove the timer itself
    RemoveEventLoopTimer(snapshotTimer);
// Clear the timer UPP
    DisposeEventLoopTimerUPP(snapshotTimerUPP);
// Clear the pointers - optional but good practice
    snapshotTimer = NULL;
    snapshotTimerUPP = NULL;
```

QuickTime and OpenGL

Getting QuickTime data out of OpenGL

- QuickTime snapshots
- QuickTime snapshot sequences
- QuickTime movies from OpenGL

QuickTime and OpenGL

QuickTime Snapshots

- Uses data from an OpenGL context to generate an image file
- Single function creates a single image file
- File type/format is arbitrary
- Let QuickTime do the work for you!

QuickTime and OpenGL

QuickTime Snapshots

- Methodology
  - **glReadPixels()** to get image data
  - New **GWorld** created with that data
  - Set up QuickTime to create an image file with that data
  - Export the data to a file

QuickTime and OpenGL

Reading pixels out of OpenGL

```
// Default behavior, but set here for clarity
    glReadBuffer(GL_BACK);

// Tell OpenGL to send the contents of the back
//  buffer
    glReadPixels(0,0,640,480,                          // Size
            GL_BGRA,                                   // Format
            GL_UNSIGNED_INT_8_8_8_8_REV,               // Type
            dataBuffer);                               // Storage

// IMPORTANT: This method swizzles the image to
// the correct orientation
    InvertGLImage(dataBuffer, imgSize, 640 * 4);
```

QuickTime and OpenGL

Creating a new GWorld

```
// Create a new GWorld with that OpenGL data
// Be sure to use NewGWorldFromPtr() to get correct  row bytes
   error = NewGWorldFromPtr(
                        &img,                       // Destination GWorld
                        k32ARGBPixelFormat,         // Pixel format
                        &glw->winRect,              // Bounding Rectangle
                        0,                          // Color table
                        0,                          // GDevice
                        0,                          // Flags
                        dataBuffer,                 // Input data
                        640*4);                     // Row bytes

// Here we would create an FSSpec for our image file (code omitted)
```

QuickTime and OpenGL

Setting up QuickTime

```
// NOTE: QuickTimeComponents.h for OSFileType definitions
// Get the default component for exporter and file type used

    err = OpenADefaultComponent(
            GraphicsExporterComponentType,
            osFileType,
            &geComp);

// Set the input GWorld to the GWorld we created
    cErr = GraphicsExportSetInputGWorld(geComp, img);

// Set the output file to the FSSpec we created previously
    cErr = GraphicsExportSetOutputFile(geComp, &imgFile);
```

QuickTime and OpenGL

Setting up QuickTime

```
// Set our compression quality if needed
    cErr = GraphicsExportSetCompressionQuality(geComp,
    codecLosslessQuality);
// Export the data
    cErr = GraphicsExportDoExport(geComp, NULL);
// Close the component
    if (geComp != NULL)
        CloseComponent(geComp);
```

QuickTime and OpenGL

QuickTime Snapshot Sequence

- Extrapolates on the previous snapshot function
  - Uses the same function call
- Creates multiple images
- File type/format again is arbitrary
- Uses timers to generate the sequence

QuickTime and OpenGL

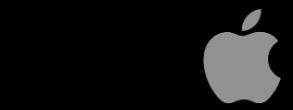
QuickTime Snapshot Sequence

```
if(seqFrameCount > 0)
{
    windowSnapshot(&mainWindow);
    seqFrameCount--;
}
else
{
    // Stop the timer from firing here
}
```

QuickTime and OpenGL

QuickTime Movies

- Captures image data from OpenGL at a specified rate
- Image data is processed asynchronously
- Creates a QuickTime movie file



QuickTime and OpenGL

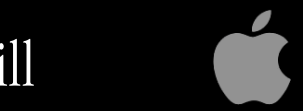
QuickTime Movies

- Methodology
  - **glReadPixels()** used to grab the back buffer and pull the contents into main memory
  - That data buffer is then placed in a buffer array for processing
  - Separate thread asynchronously processes the buffer array for use in QuickTime
  - QuickTime is then called on to build a movie with the frames in that array

QuickTime and OpenGL

QuickTime Movies

- Important caveats
  - Images need to be swizzled for correct orientation
  - Processing the image data from OpenGL needs to be done asynchronously
    - Performance will suffer otherwise
  - Arbitrary movie sizes will require more complicated array and buffer sizing
    - Frame size and frame rate will affect how much memory should be reserved
    - Dynamic allocation possible but more complex still



Demo

QuickTime and OpenGL

OpenGL Buffer Ops

Using OpenGL's Buffers

- OpenGL buffer basics
- Auxiliary buffers (AUX buffers)
- Render to texture

OpenGL Buffer Ops

OpenGL Buffers

- **GL_FRONT** and **GL_BACK**
  - Leave the front buffer alone
  - Always read/draw to the back buffer
  - You do not, under any circumstance, ever need direct access to the buffers
- **GL_DEPTH, GL_ACCUM, GL_STENCIL**
  - Depth buffer for ordering and culling
  - Accumulation buffer for composites
  - Stencil buffer for drawing containment

OpenGL Buffer Ops

OpenGL Buffers

- **GL_AUX**
  - Damage repair
  - State saving
  - Render to texture
- Damage repair
  - Blits saved content back to main context
- State saving
  - Restores previous image

OpenGL Buffer Ops

Rendering directly to a texture

- Techniques
  - Draw to an AUX buffer and use **glCopyTexImage()** or **glCopyTexSubImage** commands
    - Easy to implement
    - Still has a pixel copy
  - Use **aglSurfaceTexture()**
    - More complicated
    - No pixel copy
  - ARB_RENDER_TEXTURE
    - Not currently supported

OpenGL Buffer Ops

Rendering directly to a texture

- Using AUX buffers
  - Specify the AUX buffer you want to use with **glDrawBuffer()**
    - Ex. **glDrawBuffer(GL_AUX0);**
  - Perform any drawing you want to do
  - Set the read buffer to use the AUX buffer where you performed your drawing
    - Ex. **glReadBuffer(GL_AUX0);**

OpenGL Buffer Ops

Rendering directly to a texture

- Using AUX buffers
  - Use **glCopyTexImage2D()** for the whole buffer or **glCopyTexSubImage2D()** for only part of it to get the data you want to use

```
glCopyTexImage2D(
                    GL_TEXTURE_2D,      // Normal texture
                    0,                  // Single resolution
                    GL_RGBA,            // Internal format
                    0, 0,               // x, y start point
                    bufferW, bufferH    // width and height
                    0,);                // No border
```

OpenGL Buffer Ops

Rendering directly to a texture

- aglSurfaceTexture()

```
void aglSurfaceTexture (
            AGLContext context,                 // Main context
            GLenum target,                      // Texture target
            GLenum internalformat,              // Internal Fmt
            AGLContext surfacecontext);         // Texture ctx
```

OpenGL Buffer Ops

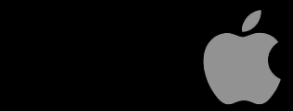
Rendering directly to a texture

- **aglSurfaceTexture()** Methodology
  - Create a p-buffer (offscreen context) to use as your texture
  - Perform drawing operations in the offscreen context
  - Call **aglSurfaceTexture()** using the offscreen context as the surfaceContext parameter

OpenGL Buffer Ops

Rendering directly to a texture

- Caveats
  - Jaguar only for **aglSurfaceTexture**
  - AUX buffer solution usable on Mac OS X only
  - AUX buffer solution still requires a pixel copy
  - **aglSurfaceTexture** does not require a pixel copy, but does require another OpenGL context



Demo

OpenGL Render to Texture

Roadmap

500 Graphics and Imaging Overview	Room A2 **Tue., 10:30am**
503 Exploring the Quartz Compositor	Hall 2 **Tue., 3:30pm**
504 OpenGL: Graphics Programmability	Room A2 **Tue., 5:00pm**
505 OpenGL: Integrated Graphics I	Room J **Wed., 9:00am**

Roadmap

506 OpenGL: Integrated Graphics II	Room J **Wed., 10:30am**
509 ColorSync and Digital Media	Room C **Wed., 5:00pm**
511 Games Solutions: Graphics, Events, and Tidbits	Room C **Thurs., 10:30am**
512 Games Solutions: NetSprocket and OpenPlay	Room C **Thurs., 2:00pm**

Roadmap

513 OpenGL: Advanced 3D	Room J **Thurs., 3:30pm**
514 OpenGL: Performance and Optimization	Room J **Thurs., 5:00pm**
516 Graphics and Imaging Performance Tuning	Hall 2 **Fri., 3:30pm**
FF018 Graphics and Imaging:	Room J1 **Fri., 5:00pm**

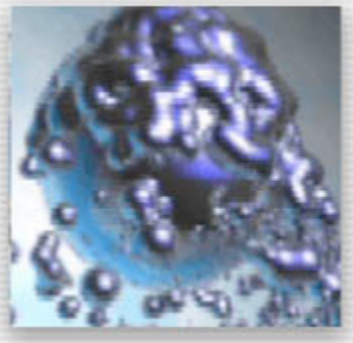
Who to Contact

Sergio Mello
3D Graphics Technology Manager
sergio@apple.com

For More Information

- O'Reilly "Learning Cocoa"
- Cocoa Developer Documentation
  http://developer.apple.com/techpubs/macosx/macosx.html
- iServices Technical Training
  http://www.apple.com/iservices/technicaltraining
- Other places
  - **www.stepwise.com**
  - **www.omnigroup.com**
  - **www.cocoadevcentral.com**

Q&A



Sergio Mello
3D Graphics Technology Manager
Worldwide Developer Relations
sergio@apple.com

http://developer.apple.com/wwdc2002/urls.html


WWDC2002


WWDC2002


WWDC2002